CPC: programming with a massive number of lightweight threads

Gabriel Kerneis, Juliusz Chroboczek

To cite this version:

Gabriel Kerneis, Juliusz Chroboczek. CPC: programming with a massive number of lightweight threads. PLACES'11, Apr 2011, Saarbrücken, Germany. pp. 30–34, 2011. <hal-00563369>

HAL Id: hal-00563369

https://hal.archives-ouvertes.fr/hal-00563369

Submitted on 4 Feb 2011

CPC: programming with a massive number of lightweight threads

Gabriel Kerneis
Université Paris Diderot
Paris, France
kerneis@pps.jussieu.fr

Juliusz Chroboczek
Université Paris Diderot
Paris, France

1 Introduction

Threads are a convenient and modular abstraction for writing concurrent programs. Unfortunately, threads, as they are usually implemented, are fairly expensive, which often forces the programmer to use a somewhat coarser concurrency structure than he would want to. The standard alternative to threads, event-loop programming, allows much lighter units of concurrency; however, event-loop programming splits the flow of control of a program into small pieces, which leads to code that is difficult to write and even harder to understand [1, 8].

Continuation Passing C (CPC) [4, 6] is a translator that converts a program written in threaded style into a program written with events and native system threads, at the programmer's choice. Threads in CPC, when compiled to events, are extremely cheap, roughly two orders of magnitude cheaper than in traditional programming systems; this encourages a somewhat unusual programming style.

Together with two undergraduate students [2], we taught ourselves how to program in CPC by writing *Hekate*, a *BitTorrent seeder*, a massively concurrent network server designed to efficiently handle tens of thousands of simultaneously connected peers. In this paper, we describe a number of programming idioms that we learnt while writing Hekate; while some of these idioms are specific to CPC, many should be applicable to other programming systems with sufficiently cheap threads.

The CPC translation process itself is described in detail elsewhere [6].

2 Cooperative CPC threads

The extremely lightweight, cooperative threads of CPC lead to a “threads are everywhere” feeling that encourages a somewhat unusual programming style.

Lightweight threads Contrary to the common model of using one thread per client, Hekate spawns at least three threads for every connecting peer: a reader, a writer, and a timeout thread. Spawning several CPC threads per client is not an issue, especially when only a few of them are active at any time, because idle CPC threads carry virtually no overhead.

The first thread reads incoming requests and manages the state of the client. The BitTorrent protocol defines two states for interested peers: “unchoked,” i.e. currently served, and “choked.” Hekate maintains 90 % of its peers in choked state, and unchokes them in a round-robin fashion.

The second thread is in charge of actually sending the chunks of data requested by the peer. It usually sleeps on a condition variable, and is woken up by the first thread when needed. Because these threads are scheduled cooperatively, the list of pending chunks is manipulated by the two threads without need for a lock.

Each read on a network interface is guarded by a timeout, and a peer that has not been involved in any activity for a period of time is disconnected. Earlier versions of Hekate which did not include this protection would end up clogged by idle peers, which prevented new peers from connecting.

In order to simplify the protocol-related code, timeouts are implemented in the buffered read function, which spawns a new timeout thread on each invocation. This temporary third thread sleeps for the

```
cps void
listening(hashtable * table) {
    /* ... */
    while(1) {
        cpc_io_wait(socket_fd, CPC_IO_IN);
        client_fd = accept(socket_fd, ...);
        cpc_spawn client(table, client_fd);
    }
}
```

Figure 1: Accepting connections and spawning threads

duration of the timeout, and aborts the I/O if it is still pending. Because most timeouts do not expire, this solution relies on the efficiency of spawning and context-switching short-lived CPC threads [4, 6].

Native and cps functions CPC threads might execute two kinds of code: *native* functions and *cps* functions (annotated with the `cps` keyword). Intuitively, cps functions are interruptible and native functions are not. From a more technical point of view, cps functions are compiled by performing a transformation to Continuation Passing Style (CPS), while native functions execute on the native stack [6].

There is a global constraint on the call graph of a CPC program: a cps function may only be called by a cps function; equivalently, a native function can only call native functions — but a cps function can call a native function. This means that at any point in time, the dynamic chain consists of a "cps stack" of cooperating functions followed by a "native stack" of regular C functions. Since context switches are forbidden in native functions, only the former needs to be saved and restored when a thread cooperates.

Figure 1 shows an example of a cps function: `listening` calls the primitive `cpc_io_wait` to wait for the file descriptor `socket_fd` to be ready, before accepting incoming connections with the native function `accept` and spawning a new thread for each of them.

3 Comparison with event-driven programming

Code readability Hekate's code is much more readable than its event-driven equivalents. Consider for instance the BitTorrent handshake, a message exchange occurring just after a connection is established. In *Transmission*[1], a popular and efficient BitTorrent client written in (mostly) event-driven style, the handshake is a complex piece of code, spanning over a thousand lines in a dedicated file. By contrast, Hekate's handshake is a single function of less than fifty lines including error handling.

While some of Transmission's complexity is explained by its support for encrypted connexions, Transmission's code is intrinsically much more messy due to the use of callbacks and a state machine to keep track of the progress of the handshake. This results in an obfuscated flow of control, scattered through a dozen of functions (excluding encryption-related functions), typical of event-driven code [1].

Expressivity Surprisingly enough, CPC threads turn out to be more expressive than native threads, and allow some idioms that are more typical of event-driven style.

A case in point: buffer allocation for reading data from the network. When a native thread performs a blocking read, it needs to allocate the buffer before the `read` system call; when many threads are blocked waiting for a read, these buffers add up to a significant amount of storage. In an event-driven program,

---

[1] `http://www.transmissionbt.com`

it is possible to delay allocating the buffer until after an event indicating that data is available has been received.

The same technique is not only possible, but actually natural in CPC: buffers in Hekate are only allocated after `cpc_io_wait` has successfully returned. This provides the reduced storage requirements of an event-driven program while retaining the linear flow of control of threads.

4 Detached threads

While cooperative, deterministically scheduled threads are less error-prone and easier to reason about than preemptive threads, there are circumstances in which native operating system threads are necessary. In traditional systems, this implies either converting the whole program to use native threads, or manually managing both kinds of threads.

A CPC thread can switch from cooperative to preemptive mode at any time by using the the `cpc_attach` primitive (inspired by FairThreads' `ft_thread_link` [3]). A cooperative thread is said to be *attached* to the default scheduler, while a preemptive one is *detached*.

The `cpc_attach` primitive takes a single argument, a scheduler, either the default event loop (for cooperative scheduling) or a thread pool (for preemptive scheduling). It returns the previous scheduler, which makes it possible to eventually restore the thread to its original state. Syntactic sugar is provided to execute a block of code in attached or detached mode (`cpc_attached`, `cpc_detached`).

Hekate is written in mostly non-blocking cooperative style; hence, Hekate's threads remain attached most of the time. There are a few situations, however, where the ability to detach a thread is needed.

Blocking OS interfaces Some operating system interfaces, like the `getaddrinfo` DNS resolver interface, may block for a long time (up to several seconds). Although there exist several libraries which implement equivalent functionality in a non-blocking manner, in CPC we simply enclose the call to the blocking interface in a `cpc_detached` block (see Figure 2a).

Figure 2b shows how `cpc_detached` is expanded by the compiler into two calls to `cpc_attach`. Note that CPC takes care to attach the thread before returning to the caller function, even though the `return` statement is inside the `cpc_detached` block.

```
cpc_detached {
    rc = getaddrinfo(name, ...)
    return rc;
}
```

(a)

```
cpc_scheduler *s =
    cpc_attach(cpc_default_threadpool);
rc = getaddrinfo(name, ...)
cpc_attach(s);
return rc;
```

(b)

Figure 2: Expansion of `cpc_detached` in terms of `cpc_attach`

Blocking library interfaces Hekate uses the *curl* library [2] to contact BitTorrent *trackers* over HTTP. Curl offers both a simple, blocking interface and a complex, non-blocking one. We decided to use the one interface that we actually understand, and therefore call the blocking interface from a detached thread.

Parallelism Detached threads make it possible to run on multiple processors or processor cores. Hekate does not use this feature, but a CPU-bound program would detach computationally intensive tasks and let the kernel schedule them on several processing units.

---

[2] `http://curl.haxx.se/libcurl/`

```
        prefetch(source, length);                          /* (1) */
        cpc_yield();                                       /* (2) */
        if(!incore(source, length)) {                      /* (3) */
            cpc_yield();                                   /* (4) */
            if(!incore(source, length)) {                  /* (5) */
                cpc_detached {                             /* (6) */
                    rc = cpc_write(fd, source, length);
                }
                goto done;
            }
        }
        rc = cpc_write(fd, source, length);                /* (7) */
    done:
        ...
```

The functions `prefetch` *and* `incore` *are thin wrappers around the* `posix_madvise` *and* `mincore` *system calls.*

Figure 3: An example of hybrid programming (non-blocking read)

5 Hybrid programming

Most realistic event-driven programs are actually *hybrid* programs [7, 9]: they consist of a large event loop, and a number of threads (this is the case, by the way, of the *Transmission* BitTorrent client mentioned above). Such blending of native threads with event-driven code is made very easy by CPC, where switching from one style to the other is a simple matter of using the `cpc_attach` primitive.

This ability is used in Hekate for dealing with disk reads. Reading from disk might block if the data is not in cache; however, if the data is already in cache, it would be wasteful to pay the cost of a detached thread. This is a significant concern for a BitTorrent seeder because the protocol allows requesting chunks in random order, making kernel readahead heuristics useless.

The actual code is shown in Figure 3: it sends a chunk of data from a memory-mapped disk file over a network socket. In this code, we first trigger an asynchronous read of the on-disk data (1), and immediately yield to threads servicing other clients (2) in order to give the kernel a chance to perform the read. When we are scheduled again, we check whether the read has completed (3); if it has, we perform a non-blocking write (7); if it hasn't, we yield one more time (4) and, if that fails again (5), delegate the work to a native thread which can block (6).

Note that this code contains a race condition: the prefetched block of data could have been swapped out before the call to `cpc_write`, which would stall Hekate until the write completes. However, our measurements show that the write never lasted more than 10 ms, which clearly indicates that the race does not happen. Note further that the call to `cpc_write` in the `cpc_detached` block (6) could be replaced by a call to `write`: we are in a native thread here, so the non-blocking wrapper is not needed. However, the CPC runtime is smart enough to detect this case, and `cpc_write` simply behaves as `write` when invoked in detached mode; for simplicity, we choose to use the CPC wrappers throughout our code.

6 Experimental results

Benchmarking a BitTorrent seeder is a difficult task because it relies either on a real-world load, which is hard to control and only provides seeder-side information, or on an artificial testbed, which might fail to accurately reproduce real-world behaviour. Our experience with Hekate in both kinds of setup shows that CPC generates efficient code, lightweight enough to run Hekate on embedded hardware. This confirms our earlier results [5], where me measured the performance of toy web servers.

Real-world workload To benchmark the ability of Hekate to sustain a real-world load, we need popular torrents with many requesting peers over a long period of time. Updates for Blizzard's game *World of Warcraft* (WoW), distributed over BitTorrent, meet those conditions: each of the millions of WoW players around the world runs a hidden BitTorrent client, and at any time many of them are looking for the latest update.

We have run an instance of Hekate seeding WoW updates without interruption for weeks. We saw up to 1,000 connected peers (800 on average) and a throughput of up to 10 MB/s (around 5 MB/s on average). Hekate never used more than 10 % of the 3.16 GHz dual core CPU of our benchmarking machine.

Stress-test on embedded hardware We have ported Hekate to *OpenWrt*[3], a Linux distribution for embedded devices. Hekate runs flawlessly on a MIPS-based router with a 266 MHz CPU, 32 MB of RAM and a 100 Mbps network card. The torrent files were kept on a USB key.

Because Hekate maps every file it serves in memory, and the MIPS routers running OpenWrt are 32-bit machines, we are restricted to no more than 2 GB of content. Our stress-test consists in 1,000 clients, requesting random chunks of a 1.2 GB torrent from a computer directly connected to the device. Hekate sustained a throughput of 2.9 MB/s. The CPU was saturated, mostly with software interrupt requests (60 % *sirq*, the usb-storage kernel module using up to 25 % of CPU).

7 Conclusions

Hekate has shown that CPC is a tool that is able to produce efficient network servers, even when used by people who do not fully understand its internals and are not specialists of network programming. While writing Hekate, we had a lot of fun exploring the somewhat unusual programming style that CPC's lightweight, hybrid threads encourage.

We have no doubt that CPC, possibly with some improvements, will turn out to be applicable to a wider range of applications than just network servers, and are looking forward to experimenting with CPU-bound distributed programs.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[2] P. Attar and Y. Canal. Réalisation d'un seeder bittorrent en CPC, June 2009. Rapport de stage.

[3] F. Boussinot. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, 2006.

[4] J. Chroboczek. Continuation-passing for C: a space-efficient implementation of concurrency. Technical report, PPS, Université Paris 7, 2005.

[5] G. Kerneis and J. Chroboczek. Are events fast? Technical report, PPS, Université Paris 7, 2009.

[6] G. Kerneis and J. Chroboczek. Continuation-Passing C, compiling threads to events through continuations. Submitted for publication, 2010.

[7] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: an efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

[8] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, 2003.

[9] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

---

[3] `http://openwrt.org`